

上周工作

讨论了上周在编写文档和代码时发现的问题：

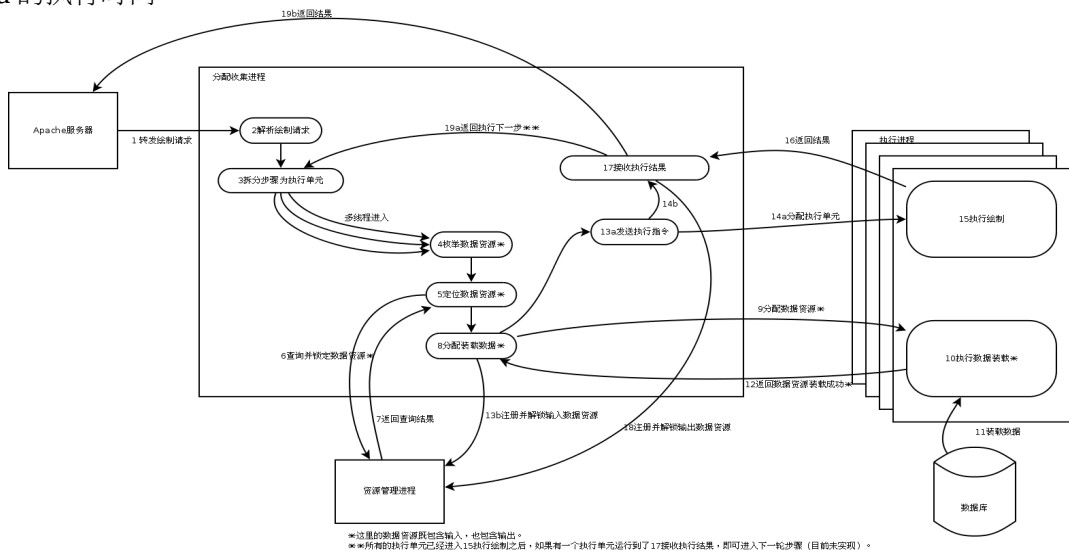
1. 讨论单机上资源管理的实现细节。按照目前的设计，为每一个请求启动一个进程，将会遇到如下两个问题：
a. 内存是基于进程分配的。如果一个绘制请求完成后，进程被关闭，那么他在内存中的数据也会被销毁，就无法重复使用。
b. OpenGL 的资源是基于线程分配的，即便是同一条进程的不同线程，也无法共享 OpenGL 资源。即便通过内存共享的方式解决了前一个问题，这个问题还是不能解决。
2. 讨论整体大资源分割细节。
3. 讨论并行端协作图细节。

根据周二讨论结果，完善协作图。

协作图的改变：将数据资源的注册和解锁分成两种情况。对于输入数据资源，可以在 13b 完成，因为他是只读的。对于输出数据则要到完成执行单元之后才能解锁，执行单元对他的访问是独占的。

将 3 生成步骤重命名为拆分步骤，即把一个步骤拆分成了几个可以并行的执行单元。

定义了 19a 的执行时间。



制订数据资源 **ID** 设计初稿。

[illegible]

数据结构:

```
union ResourceID
{
    struct
    {
        UI8 type;
        UI8 attr;
        DateTime time;
        ResourcePosition pos;
        ResourceVector res;
        UI64 param;
        UI64 reserve;
    };
    UI8 byte[1+1+sizeof(DateTime)+sizeof(ResourcePosition)+sizeof(ResourceVector)+8+8];
};
```

编写绘制请求、步骤的代码。

```
void ExecuteRequest( Request const * const request )
{
    //for each step
    for( UI32 si = 0 ; si < request->size() ; ++si )
```

```

{
    RequestStep const * const step = &request->at( si );
    IOperation * const operation = _operations.at( step->operation );
    IOperation * const exe_op = operation->NewOperation();

    //divide step into execution units
    UI32 const num_unit = exe_op->UnitSpliteStep( step );
    thread ** threads;
    IOperation ** operations;
    SAFE_NEW_ARRAY( threads , new threads * [num_unit] );
    SAFE_NEW_ARRAY( operations , new IOperation * [num_unit] );
    for( UI32 ui = 0 ; ui < num_uint ; ++ui )
    {
        operations[ui] = exe_op->NewOperation();
        SAFE_NEW( threads[ui] , new
thread( SOLSYS_BIND( &Executer::ExecuteUnit , this )
, exe_op->UnitGetOne( ui ) , operations[ui] ) );
    }

    //wait for execution unit
    for( UI32 ui = 0 ; ui < num_uint ; ++ui )
    {
        threads[ui]->join();
        SAFE_DELETE( threads[ui] );
        operation->DeleteOperation( exe_op );
    }

    //clear
    SAFE_DELETE_ARRAY( threads );
    SAFE_DELETE_ARRAY( operations );
    operation->DeleteOperation( exe_op );
}
//reply apache server
}

private:
void ExecuteUnit( RequestStep const step , IOperation * const operation )
{
    //enum required resource
    ResourceIDArray res_id;
    operation->ResourceEnum( res_id , step );

    //allocate required resource
    //deliver resource
    //register and unlock input resource

    //step once

    //gether

    //register and unlock output resource
}

```

这里对应的是分配收集进程的部分代码。协作图中 3、4、5 的代码框架已经完成。

其中无法继续往下写 6~12 是因为原有的资源管理进程使用 **MySQL** 实现，有许多局限性，主要体现在其持久化操作是多余的，速度优化困难，需要学习 **SQL** 语言，逻辑编写麻烦，多线程难以控制等。使用自己实现的资源管理进程更佳，所以开始基于 **Socket** 编写资源管理进程。

编写资源管理进程。

前面提到开始基于 Socket 编写资源管理进程，这样复杂逻辑和优化工作都是可控的了。目前已经完成了进程框架。并实现了对部分命令的响应。接下来需要实现对所有资源管理命令的响应和资源容器。

部分命令响应的代码：

```
virtual solsys::PTR ProcessMessage( solsys::Agent * const agent )
{
    solsys::BYTE buffer[MAX_BUFFER];
    solsys::PTR const size = agent->session->ValidSize();

    common::Command cmd;
    agent->session->Peek( &cmd , sizeof( cmd ) );

    switch( cmd )
    {
    case common::COMMAND_PRINT:
        {
            UI32 len;
            if( size >= sizeof( cmd ) + sizeof( len ) )
            {
                agent->session->Peek( buffer , sizeof( cmd ) + sizeof( len ) );
                solsys::PTR const len = *reinterpret_cast< solsys::PTR const *>( buffer +
sizeof( cmd ) );

                if( size >= sizeof( cmd ) + sizeof( len ) + len )
                {
                    agent->session->Read( buffer , sizeof( cmd ) + sizeof( len ) + len );
                    printf( "%s\n" , buffer + sizeof( cmd ) + sizeof( len ) );
                    return sizeof( cmd ) + sizeof( len ) + len;
                }
                else
                {
                    return 0;
                }
            }
            else
            {
                return 0;
            }
        }
        break;
    case common::COMMAND_QUERY:
        {
            if( size >= sizeof( cmd ) + sizeof( solsys::ResourceID ) )
            {
                agent->session->Read( buffer , sizeof( cmd ) + sizeof( solsys::ResourceID ) );
                solsys::ResourceID const rid = *reinterpret_cast< solsys::ResourceID const *>(
buffer + sizeof( cmd ) );

                ResourceIDMap::iterator const ite = _resources.find( rid );
                if( _resources.end() != ite )
                {
                    SI32 const num = static_cast< SI32 >( ite->second.size() );
                    if( 0 != num )
                    {
                        agent->session->Send( &num , sizeof( num ) , sizeof( num ) +
num * sizeof( solsys::MachineID ) );

                        agent->session->SendFragment( ite->second.ptr() , num *
sizeof( solsys::MachineID ) );
                    }
                }
            }
        }
    }
```

```

        else
        {
            SI32 const zero = 0;
            agent->session->Send( &zero , sizeof( zero ) , sizeof( zero ) );
        }
    }
    else
    {
        _resources[rid] = MachineIDArray();
        SI32 const locked = -1;
        agent->session->Send( &locked , sizeof( locked ) , sizeof( locked ) );
    }
    return sizeof( cmd ) + sizeof( solsys::ResourceID );
}
else
{
    return 0;
}
}
break;
case common::COMMAND_REGISTER:
case common::COMMAND_UNLOCK:
case common::COMMAND_REMOVE:
case common::COMMAND_UNKNOWN:
default:
    LOGEXCEPTION0( solsys::EXCEPTION_TYPE_FAIL );
    break;
}

return 0;
}

```

以上是 Print（调试输出）、Query 命令（查询数据资源是否存在，如果不存在则创建数据并且锁定）。还有 Register、Unlock、Remove 三个命令需要完成，分别是注册、注册解锁、删除命令。另外还有 IncreaseReference 和 DecreaseReference 两个命令，用来维护引用计数。

接下来

将 **change log** 整理成最终设计。

原有 change log 形式的设计文档难以阅读，需要整理成最终设计文档。

这样将维护两份文档：最终设计文档和 change log。每次改动最终设计文档后都将以邮件的形式发送，而 change log 将被包含在 weekly report 中。

完成资源管理进程。

- 1 完成资源容器和资源管理命令。
- 2 将现有轮询 session 来响应命令的方式，修改成利用 Socket 的 select 函数响应命令。
- 3 利用多线程响应命令。

继续编写分配收集进程。

在完成资源管理进程后，将继续编写分配收集进程的 6~12。

进度安排

目前有个粗略的进度安排:

今年的工作, 打勾(√)的表示已经完成, 划删除线(XX)的表示取消:

1. √ 底层 **Socket** 通信
2. √ 封装 **Session** 管理, 命名为 **Agent**。
3. √ 讨论单机上资源管理的实现细节。按照目前的设计, 为每一个请求启动一个进程, 将会遇到如下两个问题:
 - a. 内存是基于进程分配的。如果一个绘制请求完成后, 进程被关闭, 那么他在内存中的数据也会被销毁, 就无法重复使用。
 - b. **OpenGL** 的资源是基于线程分配的, 即便是同一条进程的不同线程, 也无法共享 **OpenGL** 资源。即便通过内存共享的方式解决了前一个问题, 这个问题还是不能解决。
4. √ 讨论整体大资源分割细节。
5. √ 讨论并行端协作图细节。
6. 9月11日讨论分配收集进程的运行细节, 以及检查光栅化程序工作进度。

目前已经决定执行进程是常驻的, 以保证内存和显存中的资源是可以被重复利用的, 所以 **MPI** 在整个框架中的作用需要重新定义。

讨论分配收集进程中各个阶段所做的工作和接口问题。

检查光栅化程序的工作进度。
7. CPU 的并行光栅化程序
8. CPU 的并行体绘制程序
9. GPU 并行光栅化程序
10. GPU 并行体绘制程序
11. √ 并行可视化的资源管理进程, 基于 **MySQL** 实现。
12. √ 编写并行可视化的资源管理进程。
13. 完善并行可视化的资源管理进程。
 - a 完成资源容器和资源管理命令。
 - b 将现有轮询 **session** 来响应命令的方式, 修改成利用 **Socket** 的 **select** 函数响应命令。
 - c 利用多线程响应命令。
14. 编写分配收集进程。
15. 编写执行进程。
16. 整理 **Change Log** 为最终设计
17. 做简单的地面、云层、海洋和地理信息绘制。
18. 做简单的地面、云层、海洋和地理信息混合绘制。

明年的工作:

19. 优化并行可视化的资源管理进程。
20. 优化边界处理。目前不对边界作冗余, 而是将涉及边界上数据的数据资源块完全装载。这样会浪费网络资源。所以应该作边界冗余。
21. 负载平衡。
22. 可靠性。
23. 复杂的地面、云层、海洋和地理信息绘制。
24. 复杂的地面、云层、海洋和地理信息混合绘制。